Group members: Eric Fields and Garrett Hill

Title: Song Structure and Lyrical Analysis of Different Songs and Genres

***Proposal/Goals Summary:***

The main goal of this project was to develop a tool used for acquiring song lyrics for a given artist and be able to determine quantitative features about different components of their songs. In particular, the tool sought to analyze things like the average number of words or lines in a variety of sections of an artists’ lyrics, such as the intros, verses, pre-choruses, and choruses. The goal was also to allow for comparative analysis between two different artists in terms of these same quantitative features of their lyrics. Additionally, the goal was to apply some sort of natural language processing techniques to the stored lyrics to train a model that could classify lyrics into a specific genre.

***What Works and What Doesn't:***

Currently, we have a fully functioning genius.com API scraper and parser stored in the song\_structures directory of our project GitHub. The main.rs file successfully scrapes lyrics from the Genius website with a passed in artists’ name and creates a corresponding directory to store the lyrics in. With a given artist passed in through the command line, song lyrics from the 10 most popular songs of the artist on the genius.com are scraped and cleaned. The stored cleaned song lyrics are then read by the tool and creates an Artist struct, which contains a Vec of another struct called Song for storing all songs of a particular artist. Each Song struct contains a vector of another struct called Structure that actually stores the lyrics of each given part of the song, such as verse or chorus. Additionally, there is a Summary struct that is created with a given Artist struct and calculates and stores many statistics about the artists’ lyrics, such as total number of words, average words per chorus, average word length, etc. A Summary struct can either be printed to the standard output or can be compared to a Summary of another Artist and printed to standard output.

The main.rs file in song\_structures allows for use with 2 modes: normal and comparison. In normal mode, the only argument passed to the command line is the name of a single artist, such as “Doja Cat”, and lyrics are scraped/cleaned and analysis only for that artist. In comparison mode, the following three commandline arguments are required: the name of the first artist, the name of the second artist, and the flag “-c”. This mode requires that the names of the artists passed in have already been run in normal mode to generate the directory and files to store their lyrics in. If their lyrics have already been stored, this mode will compare statistical quantities about their lyrics and print them to standard output.

In terms of what does not work for song\_structures, there is minimum error handling and error checking performed, so giving incorrect command line arguments could cause panicking. We ran out of time trying to add additional functionality and did not have enough time to add a lot of error checking throughout the code. We had initially planned to allow for lists of artists to be compared to each other, but coding complications arose that made it difficult to do analysis and comparisons for more than two artists at once. If we had more time, we might have been able to resolve these.

Additionally, we have a functioning analysis program called rs\_natural\_sandbox that takes in a database of key/label pairs, trains a model on the database, then guesses what label best matches a given key. There is partial functionality for directly taking in the lyrics folder of song\_structures and transforming that output into a readable database file, but we were unable to finish this part of the data transformation.

***Usage Instructions:***

The repository used for storing our code can be cloned with the following command:

git clone https://github.com/gahill18/ghef\_388z

In the ghef\_388z directory, the two main directories of interest are named song\_structures and rs\_natural\_sandbox. Dependencies for all executables are included in .toml files within the respective directories.

To run the code for song\_structures, cd into the song\_structures directory. Build the program with “cargo build”. With cargo build finished, the program can initially be called from the command line with the name of an artist passed in as a single argument. For example, if the artist you want to analyze is Lana Del Rey, you would run the following command: *cargo run “Lana Del Rey”.* If you want to include spaces in the artist name, either surround the entire name with quotation marks, or escape space chars with '\'. Running this could take a little bit of time as it requires performing a GET request to the genius website. This will store 10 lyric files in a folder with the artist's name and print out corresponding analysis to the terminal. This could be redirected to an output.txt if so desired.

To run the song\_structures code with comparison mode, make sure to have performed normal mode with at least two artists. For example, “cargo run Lizzo; cargo run Sia”. With normal mode run with at least two artists, they can be compared with the following command: “cargo run Lizzo Sia -c”. The “-c” must be the last argument and only two artists may be passed in before the -c. With this mode, analysis will be performed on both artists and printed to the terminal. This could also be outputted to an output.txt file if so desired.

To run rs\_natural\_playground, call it from the command line with “cargo run [mode] [path\_to\_dataset] [string\_to\_guess]”. You can choose either "raw" or "prefab" mode. Raw mode is supposed to read in from the lyrics folder of song\_structures' output, the location of which is specified by path\_to\_dataset, and transform this file structure into a readable database file located in rs\_natural\_sandbox/bin/output.txt. However, this functionality is not complete due to problems with escaping special characters in the lyrics text. Prefab mode takes in an already defined dataset of "[key]/[label]" pairs separated by commas. The delimiter character can be changed to something other than a comma by changing the function parameter of generate\_dataset on lines 28 and 42. The final output has no difference between modes. A statistical model will train on the database, then guess what label the passed string\_to\_guess matches best. The final string guess is printed to standard out.

***Challenges and Surprises:***

When we were implementing the song\_structures section, we did not realize that part of the Rust library we were using to interact with the Genius API no longer worked. We only figured out that it was not working through the help of our TA. This caused some major confusion for a while, as we did not consider that the library was the source of the problem. In the end, we were able to fix the problem by writing our own function to take the url returned from the library functions that did work, and use a separate Rust module to download and process the page manually.

Cleaning the lyrics returned from the genius.com website was also difficult because any text that was italicized was formatted weirdly when read in to program. This required a lot of time to go through manually and find inconsistencies between different song files and figuring out ways to correct them without affecting correctly stored lyrics. Processing the strings stored throughout the various structures used in the code in song\_structures was also a challenge with keeping up with the borrowing rules. There were many function calls that used things generated in other function calls and figuring out how to transfer ownership or keep data alive past the end of a function’s scope was challenging.

***Observations about Rust:***

Garrett’s comments:

Overall, I found the extensive community documentation and crate capabilities to be very impressive. Implementing our project proposal relied heavily on finding the right libraries for the job, which was relatively easy. However, it still irritates me on a personal level that Rust does not let me cobble together spaghetti solutions as a placeholder. It really wants me to be memory safe, which I know is the whole point of the language, but it makes me want to go back to Java, where I can break one thing to fix another.

Eric’s comments:

In terms of coding, processing all of the strings was not as easy for us as it may have been in other languages like Python. Having to deal with lifetimes and borrowing rules was difficult at times, in particular the borrowing rules through storing many structures within each other and constantly processing the strings while creating them.

The compiler for Rust is very powerful and was able to give many helpful tips throughout the project for trying to fix issues we were having. I appreciate Rust’s emphasis on checking the result of function calls that return things like Result<T> or Option<T> to make sure that the coder knows that what they are returning from a function call may not be what they wanted.